an Zucconi

Published

March 13, 2017

Tutorials
Unity
Shader
Python
Arduino
ne Learning
or Gamedev

in maths, python, tutorial

Positioning and Trilateration

Tweet Share

This post shows how it is possible to find the position of an object in space, using a technique called **trilateration**. The traditional approach to this problem relies on three measurements only. This tutorial addresses how to it is possible to take into

account more measurements to improve the precision of the final result. This algorithm is robust and can work even with inaccurate measurements.

If you are unfamiliar with the concepts of latitude and longitude, I suggest you read the first post in this series: Understanding Geographical Coordinates.

Introduction

Finding the position in space of an object is hard. What makes such a simple request so complex, is the fact that no traditional sensor is able to locate an object. What most sensors do, usually, is estimating the distance from the target object. Sonars and radars send out sound waves and calculate the distance by detecting how long it takes for their signal to bounce back. Satellites basically do the the same, bur using radio waves. Estimating the distance from an object is often relatively easy to do.

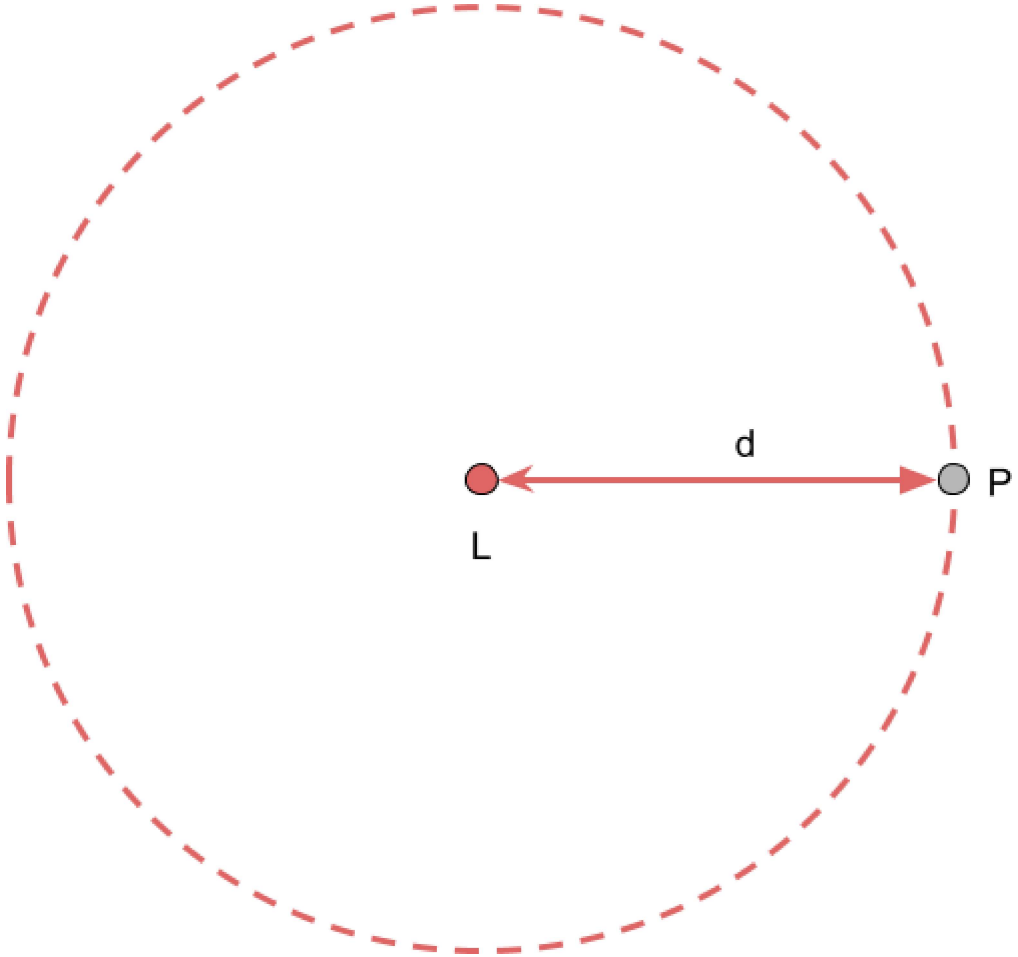
The solution to locate an object is to rely on several distance measurements, taken by different sensors (or by the same sensor at different times). **Sensor fusion** is a very active field of research, and allows to reconstruct complex information from very simple measurement. The rest of this post will introduce the concept of

trilateration, which is one of the most common techniques used to calculate the position of an object given distance measurements.

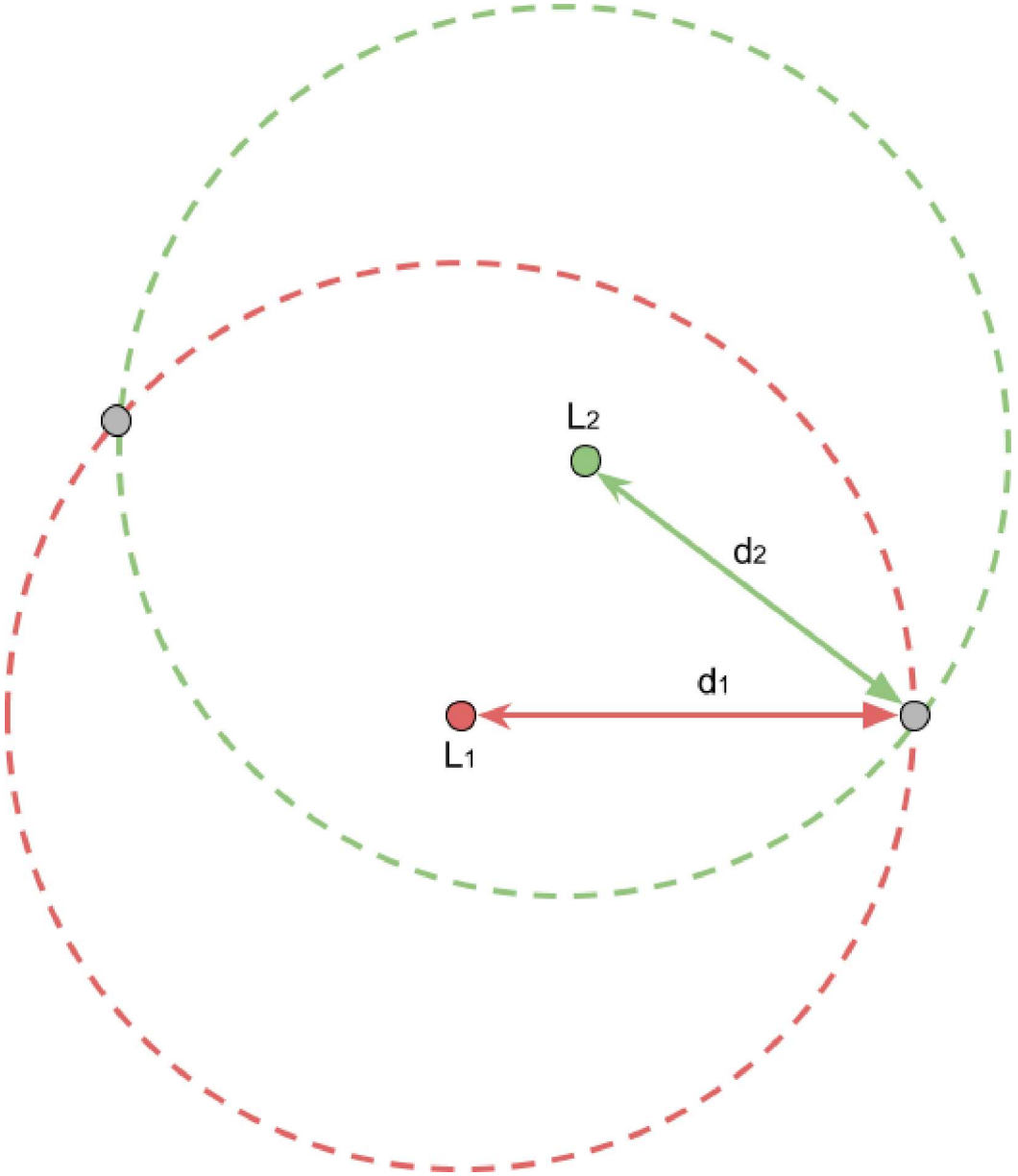
Geometrical Interpretation

Let's imagine that we want to find the location P of an object. Our first try relies on a *beacon*, or *station*, sitting at a known position L. Both P and L are expressed with latitude and longitude values. The station cannot locate P directly, but can estimate its relative distance d.

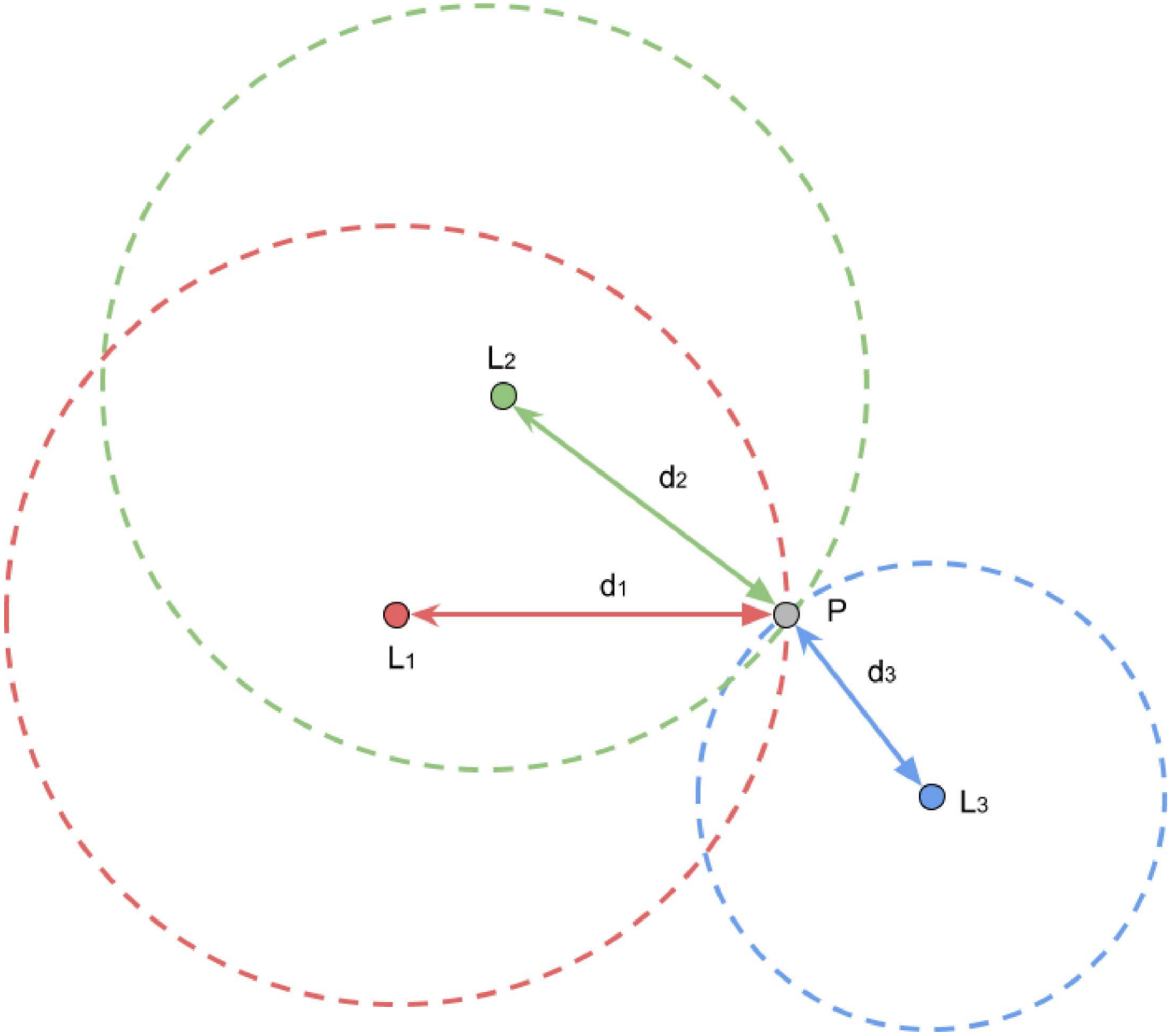
With only one station available, we cannot identify the exact position of P. What we know, however, is how *close* it is. Each point that is at distance d from L is a potential candidate for P. This means that with only one beacon, our guess of P is limited to a circle of radius d around P (below, in red).



We can improve the situation by using not one but two beacons, L_1 and L_2 (below). Our object P can only be along the circumference of the red circle. But for the same reason, it can only be along the circumference of the green circle. This means that it has to be on the intersections of the two circles. This suddenly restricts our guess to only two possible locations (in grey).



If we want to be absolutely precise, we need a third station, L_3. The three circles will meet in only one point, which is indeed the correct location of P.



In order to do that, we have a set of N *beacons* or *stations*, each one at a known locations L_i. Each beacon can sense its distance from P, which is called d_i.

Mathematical Interpretation

A point (x, y) on the Cartesian plane lies on a circle of radius r centred at (c_x, c_y) if and only if is a solution to this equation:

$$(x - c_x)^2 + (y - c_y)^2 = {d_1}^2$$

With the same reasoning, we can derive equations for the circles generated by the beacons. Each one has its own position, expressed with latitude and longitudes coordinates, (ϕ_1, λ_1), (ϕ_2, λ_2) and (ϕ_3, λ_3), respectively.

The problem of trilateration is solved mathematically by finding the point $P = (\phi, \lambda)$ that simultaneously satisfies the equations of these three circles.

$$(\phi - \phi_1)^2 + (\lambda - \lambda_1)^2 = {d_1}^2$$

$$(\phi - \phi_2)^2 + (\lambda - \lambda_2)^2 = {d_2}^2$$

$$(\phi - \phi_3)^2 + (\lambda - \lambda_3)^2 = {d_3}^2$$

There are several ways to solve these equations. If you are interested in the actual derivation, Wikipedia has a nice page explaining each step.

Optimisation Algorithm

While it is definitely true that trilateration can be seen (and solved) as a geometrical problem, this is often impractical. Relying on the mathematical modelling seen in the previous section requires us to have an incredibly high accuracy on our measurements. Worst case scenario: if the circles do not meet in a single point, the set of equations will have no solution. This leaves us with nothing.

Even assuming we do have perfect precision, the mathematical approach does not scale nicely. What is we have not three, but four points? What if we have one hundred?

The problem of trilateration can be approached from an optimisation point of view. Ignoring circles and intersections, which is the point $X = (\phi_x, \lambda_x)$ that provides us with the best approximation to the actual position P?

Given a point X, we can estimate how *well* it replaces P. We can do this simply by calculating its distance from each beacon L_i. If those distances perfectly match with their respective distances d_i, then X is indeed P. The more X deviates from these distances, the further it is assumed from P.

Under this new formulation, we can see trilateration as an optimisation problem. We need to find the pint X that minimise a certain error function. For our X, we have not one by three sources of error: one for each beacon:

$$e_1 = d_i - dist\left(X, L_1\right)$$

$$e_2 = d_2 - dist\left(X, L_2\right)$$

$$e_3 = d_3 - dist\left(X, L_3\right)$$

A very common way to merge these different contributions is to average their squares. This takes away the for possibility of negative and positive errors to cancel each others out, as squares are always positive. The quantity obtained is known as mean squared error:

$$\frac{\sum \left[d_i - dist\left(X, L_i, \right)\right]^2}{N}$$

What is really nice about this solution is that it can be used to take into account an arbitrary number of points. The piece of code below calculates the mean square error of a point `x`, given a list of `locations` and their relative `distances` from the actual target.

```
# Mean Square Error
# locations: [ (lat1, long1), ... ]
# distances: [ distance1, ... ]
def mse(x, locations, distances):
    mse = 0.0
    for location, distance in zip(locations, distances):
        distance_calculated = great_circle_distance(x[0], x[1], location[0], location[1])
        mse += math.pow(distance_calculated - distance, 2.0)
    return mse / len(data)
```

What's left now is to find the point `x` that minimises the mean square error. Luckily, `scipy` comes with several optimisation algorithms that we can use.

```
# initial_location: (lat, long)
# locations: [ (lat1, long1), ... ]
# distances: [ distance1,     ... ]
result = minimize(
    mse,                         # The error function
    initial_location,            # The initial guess
    args=(locations, distances), # Additional parameters for mse
    method='L-BFGS-B',           # The optimisation algorithm
    options={
        'ftol':1e-5,         # Tolerance
        'maxiter': 1e+7      # Maximum iterations
    })
location = result.x
```

? Can we use absolute values instead of squares?

? How many beacons should I use?

❓ Can I add multiple readings from the same beacon?

Additional Considerations

While optimisations algorithms can solve many problems, it is unrealistic to expect them to perform well if we provide little to no additional information to them. One of the most important aspect is the initial guess. For a problem this straightforward, any relatively good optimisation algorithm should converge to a reasonable solution. However, providing a good starting point could reduce its execution time significantly.

There are several initial guesses that one could use. A sensible one is to use the centre of sensor which detected the closest distance.

```
    # Initial point: the point with the closest distance
    min_distance     = float('inf')
    closest_location = None
    for member in data:
        # A new closest point!
        if member['distance'] < min_distance:
            min_distance = member['distance']
            closest_location = member['location']
    initial_location = closest_location
```

Alternatively, one could average the locations of the sensors. If they are arranged in a grid fashion, this could work well. In the end, you should test which strategy works best for you.

❓ My optimisation algorithm is not converging to an optimal. Why?

Conclusion

This post concludes the series on localisation and trilateration.

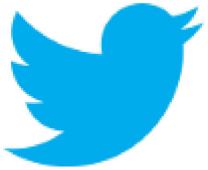
- Part 1. Understanding Geographical Coordinates
- **Part 2. Positioning and Trilateration**

Support this blog! ♥

For the past three years I've been dedicating more and more of my time to the creation of quality tutorials, mainly about *game development* and *machine learning*. If you think these posts have either helped or inspired you, please consider supporting this blog.

BECOME A PATRON

Donate



Follow @AlanZucconi

Don't miss the next tutorial!

There's a new post every Wednesday: leave your email to be notified!

Email: Address Notify me!

Write a Comment

WEBMENTIONS

Tutorial Series - Alan Zucconi
[...] Part 2. Localisation and Trilateration [...]

Understanding Geographical Coordinates - Alan Zucconi
[...] coordinates, and how they can be effectively manipulated. The second post in the series, Localisation and Trilateration, will cover the actual techniques used to identify the position of an object given independent [...]

RELATED CONTENT BY TAG GPS LOCALISATION PYTHON SCIPY TRIANGULATION TRILATERATION